理解 ThreadLocal

概述

ThreadLocal 是一种线程封闭技术，用于隔离线程间的数据，从而避免使用同步控制。

一种避免使用同步的方式就是不共享数据。如果仅在单线程内访问数据，就不需要同步。这种技术称为线程封闭。

ThreadLocal 为每条使用它的线程提供专属的内部变量。在多线程环境下访问(通过 get 或 set 方法访问)时能保证各个线程里的变量相互独立，互不影响。

基本用法

ThreadLocal 对象通常被设计为类的私有静态类型(private static)字段，用来关联线程的某种状态。

举个例子:

```
public class ThreadLocalTest {

    private static ThreadLocal<Integer> counter = new ThreadLocal<Integer>() {
        protected Integer initialValue() {
            return new Integer(0);
        }
```

```
    };

    public static class MyRunnable implements Runnable {

        private void save() {
            System.out.printf("线程[%s]保存数据，当前计数是：%s\n",
Thread.currentThread().getId(), counter.get());
        }

        public void run() {
            while(true) {
                counter.set( (counter.get() + 1) % 10 );

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.printf("线程[%s]处理业务，当前计数是：%s\n",
Thread.currentThread().getId(), counter.get());

                if(counter.get() == 0) {
                    save();
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyRunnable runnable = new MyRunnable();

        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();
```

```
    }
}
```

这个例子很简单，业务线程内有一个循环在不断的处理业务，假设每次处理业务会产生一些数据，出于性能考虑，希望每处理完 10 次业务才批量保存数据。

ThreadLocal 主要有四个方法:

- initialValue
- get
- set
- remove(例子中未使用)

下面逐一简介。

initialValue 方法

initialValue 是设计给子类重写的方法，用以返回初始化的线程内部变量。在线程第一次调用 get 时它会被调用，但如果在调用 get 之前已经调用了 set 为线程内部变量设过值，则该方法不会被调用。所以，如果你希望手动调用 set 来初始化线程内部变量,则不必重写 initialValue。

通常 initialValue 只会被调用一次，除非手动调用 remove 清除了内部变量，之后又调用 get 方法，这时 initialValue 会再被调用初始化一个新的内部变量返回。

get 方法

get 用以获取 ThreadLocal 对象关联的线程内部变量。

```
public T get()
```

set 方法

set 用以设置 ThreadLocal 对象关联的线程内部变量的值。

```
public void set(T value)
```

remove 方法

remove 用以移除 ThreadLocal 对象关联的线程内部变量，某些情况需要用它来显式地移除，以防止内存泄漏。

```
public void remove()
```

你可能会问，为什么要这么复杂，在 run 里面使用一个方法局部变量来做计数器岂不是更简单。

对于这个例子来说，确实如此，ThreadLocal 的功能性和方法局部变量没有本质的区别。

不过，ThreadLocal 相较于方法局部变量，可以帮你管理线程内部变量，降低了同一线程内多个方法和组件间传递参数的复杂度。

内部实现

如上图所示，ThreadLocal 机制主要由 Entry、ThreadLocalMap、Thread、ThreadLocal 这四个类相互协作实现的。

下面分析这四个类各自的职责和协作

Entry

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

Entry 的定义很简单，它扩展自 ThreadLocal 类型的 WeakReference 类，是一个 key-value 对类。key 是 ThreadLocal 对象的弱引用，value 是线程的内部变量。

Entry 使用弱引用作为 key 目的是，希望在外部不再需要访问 ThreadLocal 对象时可以让 GC 尽快地回收对象，而不必等到线程结束后。

当 GC 回收 ThreadLocal 对象后，再通过 Entry.get()获取 ThreadLocal 对象时返回 null，这使得内部能够感知什么时候不需要再持有对 value 的引用，从而释放 Entry 对象的引用，进而释放 value 的引用，这时如果 value 在外部没有任何引用的话(通常你不应该在外部持有对 value 的引用)，随后被 GC 回收。这种感知和释放的行为发生在 ThreadLocal 的 get、set、remove 操作时。

Thread

Thread 内部持有一个 ThreadLocalMap 类型引用的成员变量。

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;
```

threadLocals 的初始值为 null，它会延迟到初次访问时才实例化，即线程首个 ThreadLocal 对象调用 get 方法时才为 threadLocals 创建对象。

ThreadLocalMap

ThreadLocalMap 是为 ThreadLocal 而设计的 hash map，内部维护着一个哈希 table 数组，table 内保存 Entry 的对象，通过 ThreadLocal 的哈希码可索引到(哈希码需转成数组下标)。

ThreadLocal

ThreadLocal 是整个机制的总导演，对外，它提供使用的接口；对内，它协调类之间的相互协作。

ThreadLocal 内部不会持有对线程内部变量的引用，线程内部变量的引用由 Entry 对象持有，而 Entry 对象寄存在 ThreadLocalMap 内的 table 中。

每一个 ThreadLocal 对象对应一个唯一的哈希码(threadLocalHashCode)，通过这个哈希码可以从 ThreadLocalMap 中索引出对应的 Entry，从而获得线程内部变量。

这里很巧妙，ThreadLocal 对象与线程内部变量之间通过 Entry 对象间接关联，在内部只有 Entry 对象持有对 ThreadLocal 对象的弱引用，这样当外部不再使用 ThreadLocal 对象后，GC 能够回收 ThreadLocal 对象，当内部探测到 ThreadLocal 对象被回收后就接着释放 Entry 对象。

最后

Java 与 C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

通常在 Java 的世界里，我们不需要关系对象的释放，大部分情况下 GC 会自动帮我们回收。

但是如果使用 ThreadLocal 不当，是有可能导致内存泄漏的。

ThreadLocal 释放内部变量通常在以下时机:

- 线程结束后
- 显式调用 remove
- 在调用 get、set 时，如果探测到 ThreadLocal 对象的弱引用对象 get 返回 null 顺便释放。

所以，如果线程存活的生命周期很长，特别是和进程一样长的话，就要特别注意防止 ThreadLocal 引入内存泄漏的风险，在不需要再使用某个线程内部变量时记得显式调用 remove 清理掉。